# Protection traps and alternatives for memory management of an object-oriented language.*

Antony L. Hosking J. Eliot B. Moss

*Object Systems Laboratory*
Department of Computer Science
University of Massachusetts
Amherst, MA 01003

## Abstract

Many operating systems allow user programs to specify the protection level (inaccessible, read-only, read-write) of pages in their virtual memory address space, and to handle any protection violations that may occur. Such page-protection techniques have been exploited by several user-level algorithms for applications including generational garbage collection and persistent stores. Unfortunately, modern hardware has made efficient handling of page protection faults more difficult. Moreover, page-sized granularity may not match the natural granularity of a given application. In light of these problems, we reevaluate the usefulness of page-protection primitives in such applications, by comparing the performance of implementations that make use of the primitives with others that do not. Our results show that for certain applications software solutions outperform solutions that rely on page-protection or other related virtual memory primitives.

## 1 Introduction

Paged virtual memory mechanisms perform admirably when put to their intended purpose, which is to extend the address space of user programs beyond the physical memory of the machine, and for protection from other processes in multiprogrammed systems. Hardware and operating system software have been refined to achieve this sleight of hand with performance broadly acceptable to most applications.

*This work is supported by National Science Foundation Grants CCR-8658074 and CCR-9211272, Digital Equipment Corporation's Western Research Laboratory and Systems Research Center, and Sun Microsystems. The authors can be reached via Internet email addresses {hosking, moss}@cs.umass.edu.



**Proceedings 14th ACM Symposium on Operating Systems Principles Asheville, NC, Dec. 1993**

Many operating systems now allow user-level programs to exploit virtual memory mechanisms for their own purposes by providing primitives to manipulate page protections (inaccessible, read-only, read-write). User programs can also provide "handlers" to to be invoked in the event of an access violation. Using these primitives, user-level applications are able to monitor access to any of the pages in their virtual address space without explicit checks, by exploiting the paging hardware's ability to trap on access violations. As a result, application programmers have exercised their ingenuity in devising implementation solutions that make use of these virtual memory primitives. A number of these applications are enumerated by Appel and Li [3], where they argue that in light of programmers demands, designers of operating systems and hardware architectures must pay more attention to support for virtual memory primitives to make their implementations more efficient and robust.

Meanwhile, there is evidence [1] to indicate that the evolution of architectures towards pipelined RISC microprocessors, and operating systems towards micro-kernels, is making efficient implementation of these operating system primitives more difficult. As a result of this tension between the supposed demand from application programmers and the evolutionary trends of architectures and operating systems, we examine two applications cited by Appel and Li as benefiting from the availability of virtual memory primitives.

The contributions of this paper include a comprehensive performance evaluation of page-protection primitives for garbage collection and persistence, and direct comparison with corresponding software implementations. The nature of our experimental setup allows meaningful direct comparison. In addition, we project the optimal performance of applications that exploit page-protection techniques in their implementation. Our results indicate that alternative software implementations can approach, and in some cases outperform, the optimal performance of page-protection implementations.

The rest of the paper is organized as follows. In the next section we briefly describe the applications we examine, and how they are able to take advantage of virtual memory primitives. We then present the experimental setup used for gathering performance data, and our alternative implementations

of each of the applications along with their their relative performance. Finally, we summarize the major points of the paper and present our conclusions.

# 2 Applications

Appel and Li [3] describe a number of applications of virtual memory primitives, including concurrent garbage collection, shared virtual memory, concurrent checkpointing, generational garbage collection, persistent stores, extending addressability, data-compression paging, and heap overflow detection. Of these, we directly address generational garbage collection, and certain aspects of persistent stores, including object fault handling, database checkpointing, and extending addressability. Our results also have implications for the other applications, since they show that well-designed software solutions are competitive with hardware-assisted techniques.

## 2.1 Generational garbage collection

Generational garbage collectors [15, 23, 24] achieve short collection pause times partly because they separate heap-allocated objects into two or more generations and do not process all generations during each collection. Empirical studies have shown that in many programs most objects die young, so separating objects by age and focusing collection effort on the younger generations is a popular strategy. However, *any* collection scheme that processes only a small portion of the heap must somehow know or discover all pointers outside the collected area that refer to objects within the collected area. Since the areas *not* collected are generally assumed to be large, most generational collectors employ some sort of pointer tracking scheme, to avoid scanning the uncollected areas. Again, empirical studies show that in many programs, the older-to-younger pointers of interest to generational collection are rare, so avoiding scanning presumably improves performance. This is intuitively explained by the fact that newly allocated objects can only be immediately initialized to point to pre-existing (i.e., older) objects. Pointers from older generations to younger generations can be created only through assignment to pre-existing objects. Detecting such assignments requires special action at every pointer assignment to see whether that pointer must now be considered by the garbage collector when collecting the younger generations.

A number of schemes have been suggested for generating and maintaining the older-to-younger pointer information needed by generational collectors, including special-purpose hardware support [23, 24] and generation by compilers of the necessary inline code to perform the checks in software [2] (adding to the overhead of pointer stores). Ungar [23, 24] uses *remembered sets* to maintain the necessary information on a per-generation basis, recording the locations in older generations that may contain pointers into the generation. The garbage collector examines all the locations recorded in the remembered sets of the younger generations being collected to determine the live (i.e., reachable) objects.

Alternatively, dirty bits can be maintained for older generations indicating whether the generation contains pointers to objects in younger generations. The heap is divided into aligned logical regions of size $2^k$ bytes—the address of the first byte in the region will have $k$ low bits zero. These regions are called *cards* [22, 28]. Each card has a corresponding entry in a table indicating whether the card might contain a pointer of interest to the garbage collector. Mapping an address to an entry in the table involves shifting the address right by $k$ bits and using the result to index the table.

The card table can be maintained explicitly by generating code to index and dirty the corresponding table entry at every store site in the program. Alternatively, by setting the card size to correspond to the virtual memory page size, updates to clean cards can be detected using the virtual memory hardware. All clean pages in the heap are protected from writes. When a write occurs to a protected page, the trap handler records the update in the card table and unprotects the page. Subsequent writes to the now dirty page incur no further overhead. Note that *all* writes to a clean page cause a protection trap, not just those that store pointers.

The time required to determine the relevant older-to-younger pointers for garbage collection varies with the granularity of the information recorded [10]. Remembered sets have the advantage of recording just those locations that can possibly contain older-to-younger pointers. In contrast, the time to scan dirty cards is proportional to the size of the cards. While software-implemented card marking schemes are free to choose any power of two for the card size, a page trapping scheme is bound by the size of a virtual memory page. Since modern operating systems and architectures typically use a relatively large virtual memory page size (on the order of thousands of bytes), scanning overheads will be proportionally higher.

### 2.1.1 User-level dirty bits

If operating systems were to provide user-level dirty bits (as suggested by Shaw [20], and Appel and Li [3]), the overhead to reflect page traps through to the user-level protection violation handler can be avoided. Presumably, an extra user-level dirty bit would be added to each page table entry, and a system call (`dirty`) provided to return a list of pages dirtied in a given address range since the last time it was called. The system call would clear the user-level dirty bits and enable traps on the specified pages. Traps could then be handled directly in the operating system. This can have substantial savings. As reported for a MIPS R2000 [1], the time for a user program to trap to a null C routine in the *kernel* and return to the user program is 15.4$\mu$s round trip. In contrast, Appel and Li report the corresponding overhead to handle page-fault traps in *user-mode* to be 210$\mu$s on a DECstation 3100 (MIPS R2000) running Ultrix 4.1. We have confirmed

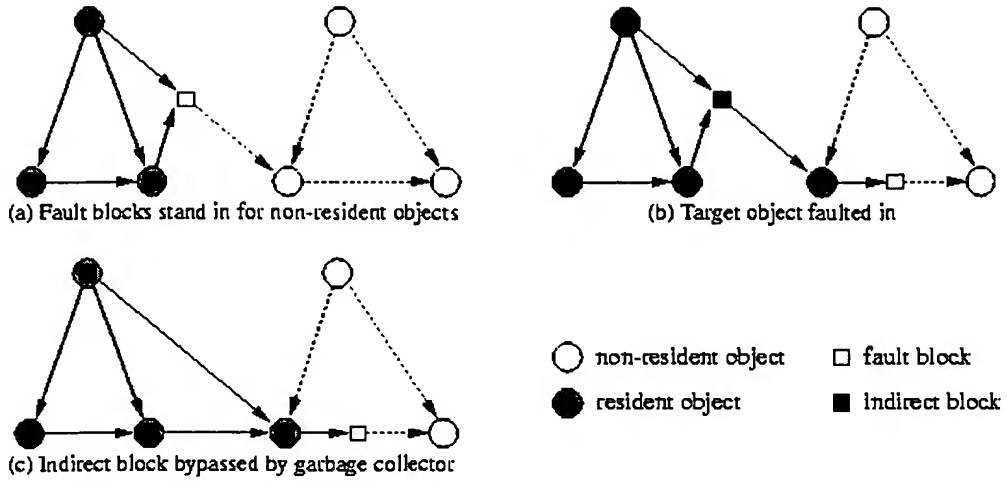


Figure 1: Fault Blocks

this with our own measurement of page traps in a tight loop using the same hardware and operating system configuration, obtaining a round-trip time of ~250$\mu$s. Note that these measurements are for a tight loop executing many repetitions, and so may tend to underestimate trap costs. Traps interspersed throughout a program's normal execution may perform less favorably, since the OS trap handling code and data structures needed to service the trap may no longer be in the hardware caches. Meanwhile, a call to `dirty` should be no more expensive than current primitives for manipulating page protections, except in copying out the dirty bit information, adding little if any extra overhead to applications that use the new primitive.

## 2.2 Persistent stores

A persistent store is a dynamic allocation heap that persists from one program invocation to the next [4, 5]. Persistent programming languages allow traversal of the data structures in a persistent store to be programmed transparently, without the need for complicated I/O or database calls to retrieve the data. Rather, the objects in the persistent store are faulted into memory on demand much as non-resident pages are automatically made resident by the virtual memory system. Moreover, a persistent program may modify the objects in the store, and commit these modifications so that their effects are permanent.

We consider three aspects in the implementation of persistence: detecting and handling object faults, extending addressability, and checkpointing of modifications.

### 2.2.1 Detecting and handling object faults

A persistent program may refer to both resident and non-resident persistent objects. Ideally, a memory-resident persistent object will be referred to by its virtual address, so that accessing the object can be as fast as accessing a non-persistent object. If the program traverses a reference to a non-resident object then it must be made available to the program in memory: we call this an *object fault*.

Tagging the references is one way to distinguish between references to resident and non-resident objects. An untagged reference is a direct memory pointer to the object in memory. A tagged reference contains an *object identifier* (OID) sufficient to locate the object on stable storage. By aligning resident objects on word boundaries, there are sufficient bits in a word for the tag. Every time a reference is traversed the tag is checked to make sure it points to a resident object; if it does not then an object fault is triggered.

An alternative is to use direct pointers for all object references, and to have resident proxy objects (we call them *fault blocks*) stand in for non-resident objects, as illustrated in Figure 1(a). A fault block contains the OID of the target object, and is tagged to distinguish it from an ordinary object. Whenever a pointer is followed, if it refers to a fault block, then an object fault is triggered. The target object is made resident and any pointers it contains are converted to direct pointers to resident objects or fault blocks. The fault block is changed to contained a tagged pointer to the now-resident object (see Figure 1(b)). We call the updated fault block an *indirect block*. If a traversed pointer refers to an indirect block then the target object can be located at the cost of an indirection. Occasional scanning (possibly by a garbage collector) can be used to bypass indirect blocks, as shown in Figure 1(c).

Object oriented programming languages can exploit the indirection implicit in the method invocation mechanism to fold residency checks into the overhead of method invocation. Method code can then directly access the fields of the object on which the method was invoked without performing further residency checks.

Rather than performing residency checks in software at each pointer traversal, fault blocks can be allocated in protected virtual memory pages so that dereferencing a pointer to a fault block is trapped. A residency check is thus implemented as a single load instruction. The protection violation handler unprotects the pages containing fault and indirect blocks, overwrites the offending fault block with an indirect block, reprotects the page, and arranges for the load instruction that caused the fault to be restarted with a direct pointer to the resident object. The fault and indirect block pages are then reprotected before resuming execution of the program.

Other approaches use page protections in a different way [14, 19, 21, 27]. When a given persistent object is to be assigned a virtual address, a page of virtual memory is reserved (although not necessarily allocated) for the page in the persistent store that contains the object. The offset of the object in the persistent page is known, allowing the virtual address of the object in the reserved virtual memory page to be calculated. Accessing the page triggers a virtual memory page trap. The trap handler reads the persistent page from the store and maps it into the previously reserved virtual page. All of the pointers in the page are then converted to direct virtual memory pointers, reserving virtual memory pages for the objects to which they refer if those objects are not already mapped into virtual memory. The faulted page is unprotected, and execution resumes. As execution proceeds, pages are reserved in a "wave-front" just ahead of the most recently faulted and swizzled pages, guaranteeing that the program will only ever see virtual memory addresses.

#### 2.2.2 Extending addressability

Persistent stores may grow so large that they contain more objects than can be addressed directly by the available hardware.[1] Dealing with this problem involves converting persistent store OIDs into virtual memory addresses, a process which has been termed *swizzling* [18]. This technique originated in early attempts to extend the address space of Smalltalk-80 [11, 12]. In any case, it relies on an OID-to-virtual-address mapping, maintained in our case by the object store software on behalf of the application.

#### 2.2.3 Database checkpointing

Modifications made to persistent data by a persistent program become permanent only when some sort of *checkpoint* operation is invoked, perhaps as the result of a database transaction commit. Given an application that modifies only a small fraction of the resident data, writing all the data back to the stable database will be hopelessly inefficient. Instead, checkpoints can *log* just those parts of the database that have been changed, allowing programs to continue execution with minimal delay. The log records can be incorporated into the database at a later time, possibly by some process running in the background. Thus, in the face of a system crash all modifications since the last checkpoint can be recovered, and the database restored to its state at that checkpoint. Generating recovery information is an important function of any persistent store, since the reliability and resilience of the database depend on it.

Detecting modifications to objects can be achieved in much the same way as for garbage collection, except that *all* updates, not just pointer stores must be recorded. Note that objects must be *unswizzled* to compare them with the object store's unmodified originals and generate log records. This is easy since we prepend OIDs to resident persistent objects. While unswizzling we may see references to new (not yet persistent) objects, which are assigned OIDs and made persistent.

## 3 Experiments

All of our experiments are based on a high-performance Smalltalk interpreter of our own design, using the abstract definition of Goldberg and Robson [7]. The implementation consists of two components: the *virtual machine* and the *virtual image*. The virtual machine implements a bytecode instruction set to which Smalltalk source code is compiled, as well as other primitive functionality. While we have retained the standard bytecode instruction set of Goldberg and Robson [7], our implementation of the virtual machine differs somewhat from their original definition to allow for more efficient execution. Our virtual machine running on the DECstation 3100 performs around three times faster than a microcoded implementation on the Xerox Dorado.

The virtual image is derived from an early commercial version of Smalltalk with minor modifications. It implements (in Smalltalk) all the functionality of a Smalltalk development environment, including editors, browsers, the bytecode compiler, and class libraries, all of which are first-class objects in the Smalltalk sense. Booting a Smalltalk environment involves loading the virtual image into memory for execution by the virtual machine.

In our persistent implementation of Smalltalk the virtual image resides in the database, and the Smalltalk environment is booted by loading that subset of the objects in the image sufficient to resume execution by the virtual machine. The bytecode instruction set is the same as in our non-persistent virtual machine, and changes to the virtual image have been minor. Rather, all extensions for persistence affected only the virtual machine, which has been augmented carefully to fault persistent objects into memory as they are needed by

[1] The recent arrival of 64-bit machines addresses this problem, but there are other good reasons to have different formats in the persistent store and in virtual memory.

the executing image. We have implemented the previously described schemes for detecting and handling object faults.

All benchmarks are coded directly in Smalltalk, and measured using a specially instrumented version of the interpreter. The instrumentation is kept constant across all implementation variants being considered, so that direct comparisons can be made—any differences in the results can only be due to the particular implementation variant being used.

## 3.1 Experimental setup

We ran our experiments on a DECstation 3100 (MIPS R2000A CPU clocked at 16.67MHz) running ULTRIX 4.1.[2] The benchmarks were run with the system in single user mode and the process's address space was locked in main memory to prevent paging. Database checkpoint operations included a call to `fsync` to force the log data to the local disk before completing. For the persistent store experiments the database was accessed remotely via NFS, with the client and server connected via a private EtherNet.

We measured elapsed time on the client machine using a custom timer board[3] having a resolution of 100 ns. The fine-grained accuracy of this timer allows separate measurement of each phase of a benchmark's execution.

All benchmarks involving random execution were made repeatable by presenting the same seed to the random number generator for each run. Random numbers are also generated before measurement of the benchmark execution, so that the elapsed times do not include the numerical computation overhead of random number generation.

## 3.2 Garbage collection

We measured three implementations of the schemes for garbage collection: remembered sets, card marking, and page traps. In contrast to our earlier performance studies [10], we have reimplemented the card and page trap schemes to avoid unnecessary scanning, by combining the precision of remembered sets with the simplicity of card marking. As the dirty cards are scanned prior to each scavenge, the older-to-younger pointers in those cards are summarized to the appropriate remembered sets, which are then used as the basis of the scavenge. The cards are thereafter treated as clean. Subsequent scavenges need only update the remembered sets by rescanning just those cards that have been dirtied since the previous scavenge.[4]

We varied the card size by multiples of four from 16 bytes up to the virtual memory page size (4K bytes). We also measured the performance of an implementation that assumes an *oracle* to discover which pages of the heap are dirty at each garbage collection. This allows us to determine the optimal performance that could be expected if a zero-cost implementation of the `dirty` operating system primitive discussed in Section 2.1 were available.

### 3.2.1 Implementation

To avoid making the remembered sets too large we record only those stores that create pointers from older objects to younger objects. This involves extra conditional overhead at every store site to perform the check, in addition to a subroutine call to update the remembered set if the condition is true. Smalltalk object references are tagged to allow direct encoding of non-pointer immediate values such as integers. Since many object references are immediate, the first action performed by the check is to filter out non-pointer stores. This is followed by a generation test to filter out "initializing" stores to objects in the youngest generation (such stores cannot create older-to-younger pointers). Finally, if the store creates a pointer from an older object to a younger object the remembered set is updated with a subroutine call. On the MIPS R2000 non-pointers are filtered in 2 cycles. Filtering initializing stores requires another 8 cycles, while filtering the remaining uninteresting stores consumes a further 8 cycles. The size of the entire inline sequence for a store typically comes to 22 instructions, including the store itself, filtering of uninteresting stores, and the call to update the remembered set; some of these are frequently skipped because of the filtering.

For the card schemes we implement the card table as a contiguous byte array, one byte per card, so as to simplify the store check.[5] By interpreting zero bytes as dirty entries and non-zero bytes as clean, a pointer store can be recorded using just a shift, index, and byte store of zero. Since the most attractive feature of card marking is the simplicity of the store check, we omit the checks used in the pure remembered set scheme to filter uninteresting stores. On the MIPS R2000 stores are recorded with just 5 instructions: 2 to load the base of the card table, a shift to determine the index, an add to index the table, and a byte store of zero. Including the store, the entire inline sequence comes to 6 instructions. If we kept the card table base in a register this sequence would shrink to 4 instructions (registers are at a premium in the interpreter). We note that the byte store instruction on the R2000 is implemented in hardware as a read-modify-write instruction, requiring several cycles for execution.

The page trap scheme requires no inline code at store sites to detect pointer stores, relying instead on the page protection hardware to trap updates to protected pages. Thus there is no longer any advantage in using a byte table to simplify the store check. Rather, it is more important that the dirty page table consume the smallest possible space. For this reason we

[2] DECstation and ULTRIX are registered trademarks of Digital Equipment Corporation. MIPS and R2000 are trademarks of MIPS Computer Systems. This version of the operating system had some official patches installed that fix bugs in the `mprotect` system call.

[3] We thank Digital Equipment Corporation's Western Research Laboratory, and Jeff Mogul in particular, for giving us the high resolution timing board and the software necessary to support it.

[4] We are indebted to one of the anonymous referees for suggesting this improvement.

[5] We first heard of this idea from Paul Wilson.

use a bit table; setting a bit indicates that the corresponding page is dirty. When a protection trap occurs the bit in the table corresponding to the modified page is set and the page unprotected.

### 3.2.2 Benchmarks

We use two benchmarks to evaluate garbage collection performance. The first is a synthetic benchmark of our own devising based on tree creation. The second consists of several iterations through the standard "macro" benchmark suite that is used to compare the relative performance of Smalltalk implementations [16]. Our benchmarks have the following characteristics:

- **Destroy**—trees with destructive updates: A large initial tree (~2M bytes) is repeatedly mutated by randomly choosing a subtree to be replaced and fully recreated. The effect is to generate large amounts of garbage, since the subtree that is destroyed is no longer reachable, while retaining the rest of the tree to the next iteration. Rebuilding the subtree causes many pointer stores, some of which create older-to-younger pointers of interest to the garbage collector. Each run performs 160 garbage collections.
- **Interactive**—10 iterations of the "macro" benchmarks: These measure a system's support for the programming activities that constitute typical interaction with the Smalltalk programming environment, such as keyboard activity, compilation of methods to bytecodes, and browsing. Each run performs 137 garbage collections.

### 3.2.3 Results

We report the elapsed time of each phase of execution of the benchmark, including:

- *running*: the time spent in the interpreter executing the program, as opposed to the garbage collector (note that running includes the cost of store checks or page traps);
- *roots*: the time spent scanning through remembered sets or card/page tables and copying the immediate survivors; [6]
- *promoted*: the time spent copying any remaining survivors; and
- *other*: the time spent in any remaining GC bookkeeping activities.

Figure 2 plots the results for the remembered set (remsets), page trap (pages), and card implementations (for card sizes of 16, 64, 256, 1024, and 4096 bytes) on the Destroy benchmark. The performance that might be obtained using a zero-cost implementation of `dirty` is estimated by taking the *running*, *roots*, and *promoted* times for the oracle-based implementation along with the *other* overheads for the card

[6] In Smalltalk the stack is stored as heap objects so there is no separate stack processing. In fact, all the process stacks are copied during each scavenge. Also, Smalltalk has only a few global variables, in the interpreter.

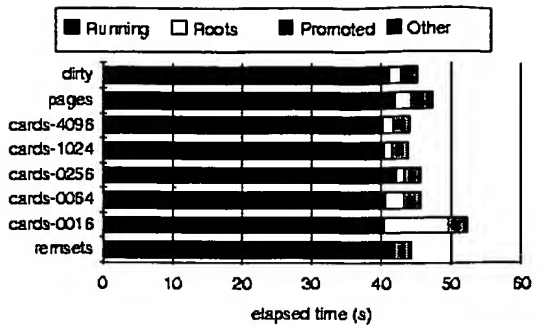


Figure 2: Destroy

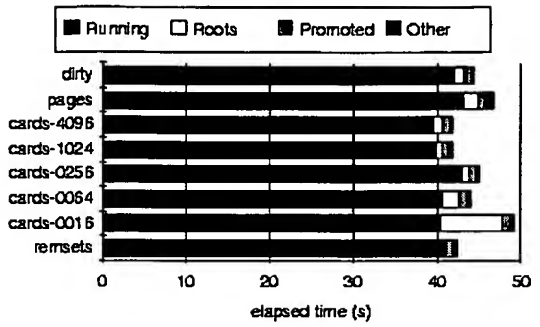


Figure 3: Interactive

marking scheme. This is plotted alongside the other measurements (dirty). The results for the Interactive benchmark are similarly shown in Figure 3.

To the extent that garbage collection overheads affect total execution time, the results are conclusive, with the page-sized granularity imposing significant overhead in scanning to determine root objects for collection. In contrast with our earlier results [10], we see that summarizing interesting pointer information into remembered sets for use in subsequent scavenges can reduce this scanning overhead, such that the card schemes are competitive with the pure remembered set scheme. Nevertheless, the pure remembered set scheme has markedly less overhead to determine the roots. Also, using a bit table versus a byte table has little effect on root processing time (the *roots* times are very similar for dirty, which scans a bit table, and cards, which scans a byte table).

The results are somewhat less conclusive for running-time overheads. The variation in running time amongst the card schemes can only be explained by hardware data cache effects (such as the specific mapping of virtual pages to physical addresses for this physically addressed cache), since the card schemes all execute the exact same code (barring differences in the shift value used to index the card table). Similarly, the fact that the oracle-based dirty scheme does not exhibit the best running time of the different implementations can only be explained as a result of such data cache effects. Neverthe-

less, dirty has running time less than pages for both benchmarks. Since these implementations use exactly the same virtual machine and garbage collection data structures, any difference is unlikely to be due to the prevously mentioned cache effects. Thus we can get some idea of the overhead to field a trap from the operating system, unprotect the appropriate page, and return to normal execution, by subtracting the running time of the oracle-based dirty scheme from that of the pages scheme, and dividing by the number of page traps. This yields a per-trap overhead of 915$\mu$s for the Destroy benchmark (864 traps), and 744$\mu$s for the Interactive benchmark (1656 traps), showing that the traps can be much more expensive than the lower bound of 250$\mu$s we obtained by measuring their cost in a tight loop. These results suggest that the frequency of traps affects their cost. Presumably, more frequent traps mean that the hardware caches are more likely to contain the operating system code and data required to service a trap, making for faster trap handling.

Given the number of store checks executed by the card schemes and the number of traps incurred by the page trap scheme for each benchmark, we can determine the trade-off between using explicit code to maintain dirty bits and a page trapping approach. Ignoring virtual-to-physical mapping cache effects, the break-even point is determined by the formula:

$$cx = fty$$

where

- $c$ = the number of store checks executed by an explicitly coded software scheme;
- $x$ = cycles per check;
- $f$ = clock frequency (16.67 MHz for DECstation 3100);
- $t$ = the number of traps incurred by a page trapping scheme;
- $y$ = $\mu$s per trap.

For these benchmarks this yields Table 1, which gives the maximum page trap overhead such that a page trapping approach will incur less running time than an alternative explicit implementation having the given overhead per store check. Let us assume that our current 5-instruction sequence for card marking executes in no more than 10 cycles. To be competitive a page trap implementation would have to incur no more than 41$\mu$s and 237$\mu$s per trap, for the Tree and Interactive benchmarks respectively. These values are significantly lower than the estimated trap overheads for these benchmarks quoted above, and lower even than the 250$\mu$s lower bound obtained for a tight loop.

We summarize the results in Tables 2 and 3, indicating the elapsed time for each of the phases as a percentage of those for dirty, and note that the total elapsed time for the 1K byte card scheme is best overall.

| | Tree | Interactive |
|---|---|---|
| Store checks ($c$) | 59646 | 654245 |
| Page traps ($t$) | 864 | 1656 |
| Cycles per check ($x$) | $\mu$s per trap ($y$) | |
| 1 | 4 | 24 |
| 2 | 8 | 47 |
| 4 | 17 | 95 |
| 5 | 21 | 118 |
| 10 | 41 | 237 |
| 15 | 62 | 355 |
| 20 | 83 | 474 |
| 50 | 207 | 1185 |
| 100 | 414 | 2370 |
| 150 | 621 | 3555 |
| 200 | 828 | 4740 |

Table 1: Break-even points for GC implementations that use page trapping vs explicit checks

## 3.3 Object fault handling

Persistent Smalltalk is obtained by extending the virtual machine to handle persistence. No modifications have been made to the bytecode instruction set to support persistence, and changes to the virtual image have been minor. Rather, all extensions involve the virtual machine, so that objects are faulted as they are needed by the executing image.

Permanent storage for the virtual image is provided by an underlying persistent object storage manager [17]. Since objects are too small a unit for efficient individual transfer to and from disk, the storage manager groups objects together into *physical segments* for transfer between the permanent database and its in-memory buffers. Physical segments may have arbitrary size (up to some large system-defined limit). Thus a physical segment may contain any number of objects. Objects within a physical segment are further grouped into *logical segments* (of at most 255 objects) for efficient management of the OID space (objects in a given logical segment have the same high bits in their OID). Applications can take advantage of these groupings to cluster related objects for retrieval.

### 3.3.1 Implementation

We have implemented two variants of the fault block approach to detecting non-residency.[7] The first uses explicit software residency checks in the virtual machine while the other exploits page-protection.

Computation in Smalltalk proceeds by sending *messages* to objects. The effect of sending a message is to invoke a *method* on the receiver of the message. Invoking a method

[7] We also implemented a reference tagging scheme, but it was clearly uncompetitive.

| Phase | remsets | cards-0016 | cards-0064 | cards-0256 | cards-1024 | cards-4096 | pages |
|---|---|---|---|---|---|---|---|
| Running | 101% | 98% | 99% | 103% | 98% | 98% | 102% |
| Roots | 19% | 542% | 153% | 64% | 58% | 93% | 138% |
| Promoted | 97% | 107% | 102% | 100% | 99% | 100% | 101% |
| Other | 94% | 107% | 104% | 96% | 96% | 100% | 360% |
| Total | 98% | 115% | 101% | 101% | 97% | 98% | 105% |

Table 2: Phases of execution for Destroy as percentage of dirty

| Phase | remsets | cards-0016 | cards-0064 | cards-0256 | cards-1024 | cards-4096 | pages |
|---|---|---|---|---|---|---|---|
| Running | 98% | 97% | 97% | 102% | 95% | 94% | 103% |
| Roots | 22% | 605% | 173% | 71% | 56% | 95% | 141% |
| Promoted | 100% | 110% | 103% | 101% | 99% | 100% | 100% |
| Other | 96% | 106% | 104% | 98% | 97% | 100% | 255% |
| Total | 95% | 111% | 99% | 102% | 94% | 94% | 105% |

Table 3: Phases of execution for Interactive as percentage of dirty

may be thought of as a procedure call. Precisely which method is invoked depends on the *class* of the receiver, so every Smalltalk object contains a pointer to its class, which is itself a Smalltalk object. Because computation is driven by the sending of messages, most objects will become resident only when a message is sent to them. By arranging for fault and indirect blocks to respond to messages by forwarding the message to their target object (faulting the object as necessary), message sends to resident objects typically incur no extra overhead.

Byte-compiled methods (code) and stack frames are also first-class objects in Smalltalk. By making further constraints on the residency of certain references contained in these objects we are able to restrict all residency checking to method invocation; even there the overhead is typically incurred only when a method is invoked on a non-resident object.[8]

The effect of the residency constraints is felt whenever an object is made resident. Pointer fields that are subject to a residency constraint must be swizzled to refer directly to their target. Unconstrained pointer fields are swizzled to point directly to their target only if the target object is already resident—the storage manager supports the efficient mapping of OIDs to resident objects. Otherwise, the pointer field is swizzled to refer to a fault block.

Since there may be multiple references to a given fault block dispersed through the registers and memory of the virtual machine, we arrange for object faults to bypass the indirection that would otherwise be created when a fault block is converted to an indirect block. This is not strictly necessary for the software fault detection scheme, since traversing a pointer to an indirect block can be quickly handled at the cost of an indirection—for a fault block, the full object fault mechanism must be invoked to translate the OID contained in the fault block. However, for a page trapping scheme, the overhead of the traps is high enough to justify expending some effort to eliminate references to (ex-fault) indirect blocks, to avoid repeated loading and faulting on those references. To support this, each page of allocated fault blocks has a remembered set associated with it, recording all persistent objects whose pointer fields have been swizzled to refer to fault blocks that lies in the page. At each object fault we scan the objects in the remembered set to update any pointers and bypass the indirection. For a fair comparison with hardware-assisted variants we also apply this indirection elimination to the software scheme.

The architecture leaves open the possibility of making any number of objects resident at one time. In an earlier study [9] we considered the granularities inherent in the underlying object storage manager: individual objects, logical segments, and physical segments. Swizzling just one object at a time has the advantage of faulting just those objects needed by the program for it to continue execution. Swizzling an entire logical or physical segment at a time allows the program to take advantage of any clustering present in the physical layout of objects in the database. Also, since all the objects in a segment can be mapped before they are swizzled, any intra-segment references can be converted to direct pointers. If the static clustering is a good approximation to the dynamic locality of access by the program, then the speed of program execution will improve since fewer object faults will occur.

For these experiments we swizzle entire logical segments, and compare several versions of the virtual machine that differ only in their implementation of fault detection, and

[8] Primitives may need to perform additional residency checks if they access objects other than those whose residency is guaranteed by the constraints.

| Scheme | Description |
|---|---|
| non-persistent | Non-persistent, unadorned with residency checks |
| FB-resident | Non-persistent, augmented with software fault block residency checks |
| PF-resident | Non-persistent, augmented with the page trap handling code, plus necessary support to decode load instructions that might cause a trap |
| FB | Persistent, fault blocks, swizzle 1 logical segment at a time |
| PF | Persistent, fault blocks allocated in protected pages, swizzle 1 logical segment at a time |

Table 4: Fault detection schemes measured in experiments

whether they are running against a completely resident virtual image (non-persistent Smalltalk) or against an image that is faulted on demand (persistent Smalltalk). Table 4 enumerates the variants.

### 3.3.2 Benchmarks

We use the Lookup and Traversal portions of the OO1 object operations benchmarks [6]. The OO1 benchmark database consists of a collection of 20,000 "part" objects, indexed by part numbers in the range 1 through 20,000, with exactly three "connections" from each part to other parts. The connections are randomly selected to produce some locality of reference: 90% of the connections are to the "closest" 1% of parts, with the remainder being made to any randomly chosen part. Closeness is defined as parts with the numerically closest part numbers. The part database and the benchmarks are implemented entirely in Smalltalk, including the B-tree used to index the parts.

The benchmarks operate as follows:

- **Lookup** fetches 1,000 randomly chosen parts from the database. For each part a null procedure is invoked, taking as its arguments the *x*, *y*, and *type* fields of the part.
- **Traversal** fetches all parts connected to a randomly chosen part, or to any part connected to it, up to seven hops (for a total of 3,280 parts, with possible duplicates). Similarly to the Lookup benchmark, a null procedure is invoked for each part, taking as its arguments the *x*, *y*, and *type* fields of the part.

Each measure is typically run 10 times, the first when the system is cold, with none of the database cached (apart from any schema or system information necessary to initialize the system). Each successive iteration fetches a ***different*** set of random parts. Before the first run of each series of benchmark iterations a "chill" program is executed on the client to ensure that the operating system file buffers of both client and server have been flushed of all database segments, so that the first iteration is truly cold.

In addition to the ten cold-warm iterations, we measured the elapsed time for a *hot* iteration of the Traversal benchmark, by beginning at the same initial part used in the last of the warm iterations. This hot run is guaranteed to traverse only resident objects, and so will be free of any overheads due to swizzling and retrieval of non-resident objects.

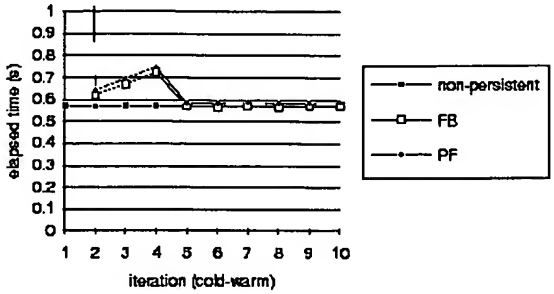


Figure 4: Lookup

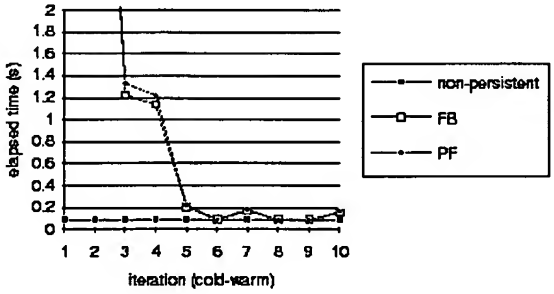


Figure 5: Traversal

### 3.3.3 Results

The elapsed times for the cold-warm iterations of each benchmark are plotted in Figures 4 and 5, expanding the scale to focus on the warm performance, with the non-persistent performance as a baseline. The FB and PF schemes behave very similarly, with warm performance close to optimal. However, the software-mediated FB scheme has better performance overall.

We summarize the benchmark results in Table 5, reporting the average elapsed time (in seconds) of the 10 iterations for the non-persistent variants (since the database is

| Scheme | Lookup Average | | Traversal Average | | |
|---|---|---|---|---|---|
| non-persistent | 0.57 | | 0.089 | | |
| FB-resident | 0.57 | | 0.089 | | |
| PF-resident | 0.57 | | 0.090 | | |
| | Cold | Warm | Cold | Warm | Hot |
| FB | 28.11 | 0.57 | 14.79 | 0.154 | 0.088 |
| PF | 29.29 | 0.59 | 15.34 | 0.162 | 0.089 |

Table 5: Elapsed times for object faulting benchmarks (seconds)

| Scheme | Lookup | | | Traversal | | |
|---|---|---|---|---|---|---|
| | $a$ | $b$ | $r$ | $a$ | $b$ | $r$ |
| FB | 0.568 | 0.000214 | 0.9015 | 0.090 | 0.000181 | 0.9885 |
| PF | 0.585 | 0.005967 | 0.9996 | 0.102 | 0.004369 | 0.9805 |

Table 6: Fault overheads (seconds)

always resident and warm), and cold (first iteration), warm (10th iteration), and hot times for the persistent variants. The non-persistent variants exhibit little difference in their performance, indicating that the overhead of the run-time residency checks is slight.

To get some sense of the cost of the page traps and software-mediated object faults we have obtained linear regression fits of the running time (time spent in the interpreter actually executing bytecodes as opposed to swizzling, bypassing indirections, or reading from disk) versus the number of faults occurring for each iteration of the benchmark. The model used is:

$$y = a + bx$$

where

- $y$ = running time (excluding swizzling and other fault handling overheads);
- $a$ = y-axis (running time) intercept;
- $b$ = seconds per fault;
- $x$ = number of faults.

The fits obtained are good and the coefficients are given in Table 6, as well as the linear correlation coefficient $r$. The $b$ coefficient is a measure of the number of seconds required to get in and out of the object fault handler, either through software checks to detect faults or through a protection trap handler. The results for PF show that trap handling overhead is once again much greater than the 250$\mu$s value obtained when measured for a tightly coded loop, but the high per-trap cost is not unreasonable considering that each fault involves substantial work to eliminate references to the trapped fault block, which will significantly disturb the state of the hardware caches. Upon resumption of normal execution the hardware instruction and data caches must be reloaded before peak execution speeds can be achieved. The results for FB reveal just how fast protection traps have to be in order to outstrip the software implementation—software-mediated fault detection overheads are less than 250$\mu$s for both benchmarks.

To summarize, we have shown that software object faulting schemes can be made to have performance close to optimal. On the other hand, page trap schemes can be significantly slowed by the cost of the traps. While we cannot vouch for the performance of a direct-mapped scheme in the style of Texas and ObjectStore, the fact that software object faulting can give performance close to optimal makes it difficult to beat.

### 3.4 Database checkpointing

A checkpoint operation consists of copying and unswizzling modified and newly-created objects (or modified subranges of objects) back to the storage manager's buffers for eventual return to the stable database, along with generating a log record describing the range and values of the modified region of the object. The log record is generated by comparing the old and new versions of the object as it is unswizzled—we generate a difference log indicating the changes made to the object. Unswizzling may encounter pointers to objects newly-created since the last checkpoint. These objects must be assigned an OID and unswizzled in turn, perhaps dragging further newly-created objects into the database.

### 3.4.1 Implementation

We have implemented and measured four schemes for detecting updates to persistent objects. The first uses a remembered set to record persistent objects that have been modified since the last checkpoint. To keep the remembered set from becoming too large we record only updates to persistent objects. This requires a check to see that the updated object is located in the separately managed persistent area of the volatile heap. If the updated object is persistent then a subroutine is invoked to enter the object's pointer in the remembered set. On the MIPS R2000, non-persistent objects are filtered in 9 cycles. The entire inline sequence is 12 instructions long.

Rather than noting updated objects in a remembered set, the second scheme marks objects as they are updated by setting a bit in the header of the object when it is modified. Upon checkpoint all resident persistent objects must be scanned to find those that have been updated.

Card marking can also be used to track updates for log generation. We compare card schemes that use card sizes of 16, 64, 256, 1024, and 4096 bytes, as well as a fourth approach which uses page protections to monitor updates. These schemes are implemented exactly the same as for the garbage collector store checks. The card schemes use a byte table and 5 instructions to dirty a card. The page protection scheme uses a bit table and traps updates to protected pages.

For small objects the remembered set scheme is ideal. However, updates to large objects may suffer from poor locality with respect to the object size, resulting in unnecessary unswizzling upon checkpoint. Thus checkpoint overheads are bounded solely by the size of the object. The object marking scheme suffers from the need to examine every resident persistent object to find those that have been modified. If only a few objects have been modified then it must examine many more objects than need to be unswizzled. The card and page protection schemes record updates based on fixed-size units of the address space. Similarly to garbage collection, we can expect the size of the cards to influence checkpoint costs, since large cards imply higher unswizzling overheads.

These benchmarks all use the exact same object faulting and swizzling scheme, while varying the update detection mechanism.

### 3.4.2 Benchmarks

Previous studies have extended the Traversal operation of the OO1 object database benchmarks to also perform some modification of part objects [26]. Each part accessed during the traversal may be updated based on some known probability fixed in advance. For example, if the probability of update is 0.5 then approximately half of all parts visited will be modified. The update consists of incrementing the $x$ and $y$ 4-byte integer fields of the part. A checkpoint operation is performed at the end of each traversal to commit the changes to the database.

In order to best understand the behavior of the update detection mechanisms in the absence of other effects such as object faults and swizzling, we measured the time to run 10 *hot* iterations of the update benchmarks, by beginning each hot iteration at the same initial part used in the last of the warm iterations. These hot runs are guaranteed to traverse only resident objects, and so will be free of any overheads due to swizzling and retrieval of non-resident objects.

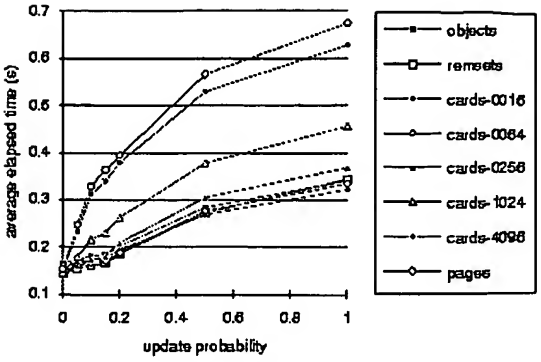


Figure 6: Hot update

### 3.4.3 Results

Figure 6 summarizes the average elapsed time for the ten hot iterations at each of the update probabilities. The results are clearer when we break the total elapsed time down into separate phases of execution, as plotted in Figures 7–10. *Running* is the time spent in the virtual machine executing the bytecodes of the benchmark program, including the overhead to note modifications. The checkpoint operation itself is decomposed into:

- *old*: time to locate and unswizzle old modified objects and generate log entries for them;
- *new*: time to unswizzle new persistent objects and generate log entries for them;
- *write*: time to flush the log records to disk;
- *other*: time to perform other bookkeeping, such as managing page protections.

For update probability $p = 0$, there are very few old modified objects to be unswizzled, so the remembered set scheme shows little overhead for this phase. The card and page schemes incur overhead to scan the card table. Since the card table is larger for smaller cards, the overheads are correspondingly higher, while the object marking scheme must scan all the objects for very little gain. Note that every checkpoint also generates a small number of new persistent objects—recall that stack frames are objects and may persist, so the checkpoint must log any newly-allocated active stack frames, to allow resumption of execution from the checkpoint in the case of a crash.